Problem Set 1: Solutions

1. Scripto Continua

a. Suppose maximizing our utility corresponds to minimizing the number of words in the output segmentation. Construct a deterministic state space model for this task.

We can model this problem with states that represent "remaining characters that still need to be segmented". In the initial state, the entire input needs to be segmented. For example, if the input sequence is "themeterman",

s_{start} = ["themeterman"]

In this model, actions will represent choosing the next word to segment. For any state s, Actions(s) is the set of dictionary words that we could choose as our next segment.

Actions(s) = {w: w in D and w is a prefix of the remaining input}

In our example,
Actions(s_{start}) = {"the", "them", "theme"}

Taking an action (choosing the next word to segment) removes that word from the input sequence. For example,

Succ(s_{start}, "the") = ["meterman"]
Succ(s_{start}, "them") = ["eterman"]
Succ(["terman"], "term") = ["an"]

All actions have the same cost since we are only interested in minimizing the number of words in the output segmentation.

Cost(s, a) = 1

Finally, we have reached the goal when no input remains to be sequenced.

IsGoal(s) = **1**{s has no remaining input}

b. What search algorithms (out of the following, BFS, DFS, UCS, A*, Bellman-Ford) would produce a minimum cost path for your model and why?

BFS	yes, edge costs are all equal, so shortest path = minimum cost path
DFS	no, not optimal in general.
UCS	yes
A*	yes
Bellman-Ford	yes

c. If our goal is to maximize the number of words in the segmentation, revise the state space model from above. Which search algorithms work now?

One simple way to revise the model is to change the cost associated with each action.

Cost(*s*, *a*) = -1

In other words, when adding a word to the output segmentation we must now "pay" a cost of -1 (i.e. we get a reward of +1). The minimum-cost path in this case is the longest path, which is exactly what we want.

Unfortunately, our new model has negative edge costs, which doesn't work with most of our algorithms:

BFS	no
DFS	no
UCS	no
A*	no
Bellman-Ford	yes

No negative-weight cycles: the state/action graph is acyclic since every action leads to a new state with less remaining input.

d. Fluency(w1, w2). Modify the state space model from above to find the most fluent segmentation.

We can modify states to keep track of not only the remaining input but also the most-recently segmented word, i.e. *s* = [input, last_word]. For example,

s_{start} = ["themeterman", *null*]

Succ(s_{start}, "the") = ["meterman", "the"]
Succ(s_{start}, "them") = ["eterman", "them"]
Succ(["meterman", "the"], "met") = ["erman", "met"]

Now, the cost of segmenting a word can incorporate fluency:

Cost([*s*, *null], w_2) = *0*
Cost([*s*,w_1], w_2) = -Fluency(w_1, w_2)

We negate Fluency(w_1, w_2) because a *high* fluency should correspond to a *low* cost.

2. Searchable Maps

a. Define a consistent heuristic for a search for the fastest path from s to t based on their Haversine distance.

Our heuristic must underestimate the true time needed to get from a node *n* to *t*. In order to accomplish this, we will assume that we can drive in a straight line from *n* to *t* (along the earth) at maximum speed. The time needed to travel from *n* to *t* under these conditions is:

$$h(n) = \frac{1}{S_H} G(n, t)$$

b. Use either the triangle inequality rule or the reverse triangle inequality rule in conjunction with landmark travel time to formulate a consistent heuristic $h_L(n)$ for a node n expanded by an A* search between s and t.

Using the reverse triangle inequality,
$h_L(n) = |T(n, L) - T(L, t)|$
[This heuristic is valid since all roads are traversable in both directions.]

c. Let h_1 and h_2 be consistent heuristics. Define a new heuristic $h(s)=\max\{h_1(s), h_2(s)\}$. Prove that h is consistent.

Since h_1 and h_2 are consistent,

$$h_1(n) \leq c(n, a, n') + h_1(n')$$

$$h_2(n) \leq c(n, a, n') + h_2(n')$$

Since $h_1(n') \leq h(n')$ and $h_2(n') \leq h(n')$ [by definition], we can make a simple substitution to get

$$h_1(n) \leq c(n, a, n') + h(n')$$

$$h_2(n) \leq c(n, a, n') + h(n')$$

Since either $h_1(n)$ or $h_2(n)$ must equal $max(h_1(n), h_2(n))$, in both cases we can substitute into one of the equations above to get

$$max(h_1(n), h_2(n)) \leq c(n, a, n') + h(n')$$

Equivalently,

$$h(n) \leq c(n, a, n') + h(n')$$

Therefore, *h* meets the requirements of a consistent heuristic.

d. Use the intuition from part (c), and your heuristics from parts (a) and (b) to create a single fastest path heuristic function.

We can combine the heuristic from part (a) with the L heuristics from part (b):

$h(n) = max(h_1(n),\ h_2(n),\ \dots\ ,\ h_L(n),\ \frac{1}{S_H}G(n,t))$

Intuitively, taking a *max* allows us to choose the best heuristic out of the set.

e. If new edges are added to the search space graph, does h remain consistent for the new model? What if edges are removed? In both cases, give a proof or a counterexample.

Adding new edges:

No, it is not necessarily consistent. In the extreme case, suppose *h* is a "perfect" heuristic for driving directions, i.e. $h(n)$ gives the true minimal cost of getting from *n* to *t* (on the old map). [This is a perfectly valid heuristic... just a really really good one]

This route might contain some slow, winding roads, so we might want to add a new highway directly from *n* to *t*. In this case, our old heuristic will *overestimate* the cost of getting from *n* to *t* in the new graph, so it cannot be consistent.

Removing edges:

Yes, it remains consistent. The consistency requirement can be thought of as a set of inequalities that *h* must satisfy, one for each edge in the graph:

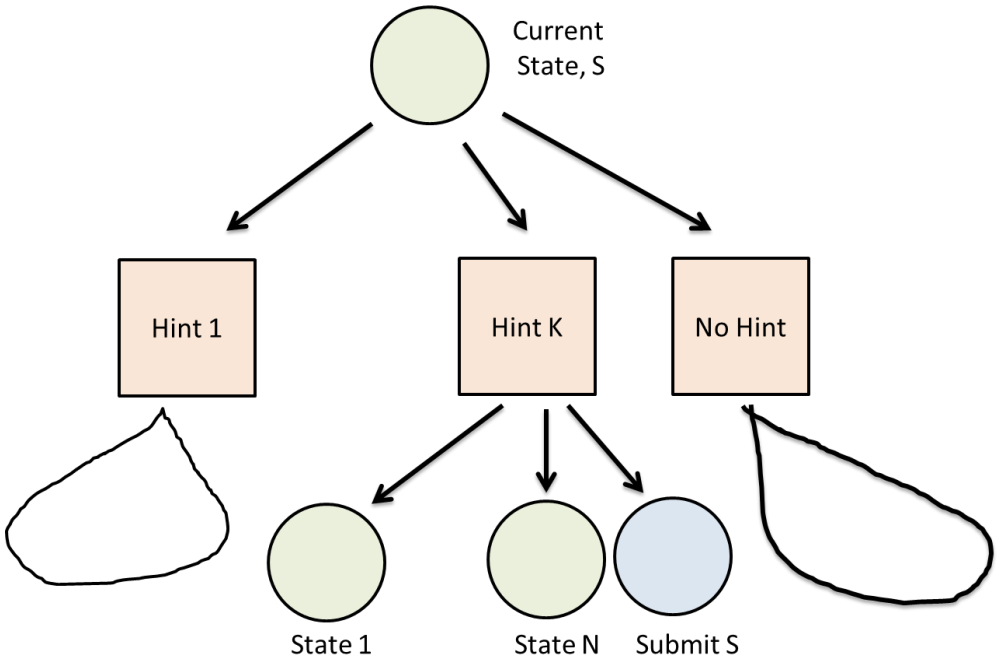
For every edge (*n*, a, *n*') in the graph, *h* satisfies $h(n) \leq c(n, a, n') + h(n')$.

Removing edges from the graph does not change whether these inequalities hold: *h* hasn't changed, and c(n, a, n') remains the same for any edge left in the graph. Therefore, the statement above continues to apply for every edge in the modified graph, and so *h* remains consistent.

Shortest path between two nodes with fewer edges is a **relaxed** problem.

3. Autonomous TA

Here is a very basic solution.



State:

- Current programming state
- How many time periods the student has been working
- Which hints have been given

Actions:

- Give any of the hints
- Give no hint

Successors T(s, a):

- Each of the states that a student can be in with time = s.time + 1
- A submission state

Start State:

- The current student code (note: not the starter code)

Reward / Cost of an action:

- None

Terminal State / Check:

- If the state is a submission

- If the time period has passed a certain limit (to prevent trees of infinite depth)

Terminal Utility:

- Expected midterm grade